

## UNIT -3

### 1Q. Explain the need of NoSQL databases. Compare SQL vs NoSQL with examples.

#### Answer:

Traditional databases followed the **Relational Database Management System (RDBMS)** model, which uses tables, rows, and fixed schemas. As modern web applications grew in size and complexity, handling **big data**, **high traffic**, and **unstructured data** became difficult using SQL databases. To overcome these limitations, **NoSQL databases** were introduced.

While SQL databases such as MySQL, Oracle, and PostgreSQL have been widely used for decades, the emergence of **modern web applications** has exposed several limitations of the relational model. The rapid growth of **web-scale applications**, **big data**, **social media platforms**, and **real-time systems** created a need for a new type of database system known as **NoSQL databases**.

#### Traditional SQL Databases Became Insufficient:

##### 1. Explosion of Data Volume

Modern applications generate huge volumes of data from:

- User interactions
- Mobile devices
- IoT sensors
- Social networks

Relational databases were not designed to efficiently handle **massive distributed datasets**.

##### 2. Rigid Schema Structure

SQL databases require a **predefined schema**:

- All rows must follow the same structure
- Any schema change requires table alteration

**Frequently changing data models** make SQL databases less flexible for modern applications.

##### 3. Scalability Issues

SQL databases mainly support **vertical scaling**:

- Increasing CPU, RAM, or storage

This approach is:

- Expensive
- Limited by hardware capacity

In contrast, modern systems demand **horizontal scalability**.

##### 4. Performance Bottlenecks

Complex joins, normalization, and transactional overhead slow down performance when handling:

- High read/write traffic
- Real-time requests

### Need for NoSQL Databases

NoSQL databases were introduced to solve the above limitations.

**NoSQL (Not Only SQL)** databases are designed to handle **large-scale, distributed, schema-less data** with high availability and performance.

## **Key Reasons for Using NoSQL Databases:**

### **1. Schema Flexibility**

NoSQL databases do not require fixed schemas.

Each document can have different fields.

✓ This matches real-world application data that evolves over time.

### **2. Horizontal Scalability**

NoSQL databases scale by adding more servers instead of upgrading a single server.

✓ This makes them suitable for cloud and distributed systems.

### **3. High Performance**

NoSQL databases reduce:

- Join operations
- Complex relational queries

✓ Data is often stored together, improving read speed.

### **4. Distributed Architecture**

NoSQL databases are **designed for distributed systems**, supporting:

- Replication
- Sharding
- Fault tolerance

### **5. Natural Fit for JavaScript Applications**

MongoDB stores data in **JSON-like documents**, which closely resemble JavaScript objects.

✓ This makes MongoDB ideal for **Node.js applications**.

## **Comparison: SQL vs NoSQL Databases**

Feature	SQL Databases	NoSQL Databases
Data Model	Tables (Rows & Columns)	Documents / Key-Value
Schema	Fixed	Dynamic
Scalability	Vertical	Horizontal
Transactions	ACID	BASE
Joins	Supported	Rare / Not Required
Data Structure	Normalized	Denormalized
Performance	Slower for big data	Faster for large datasets
Examples	MySQL, Oracle	MongoDB, Cassandra

## **Example: SQL Database**

### **Student Table (Relational Model)**

```
CREATE TABLE Student (  
  id INT PRIMARY KEY,  
  name VARCHAR(50),  
  branch VARCHAR(10),  
  marks INT  
);
```

✓ All rows must strictly follow this structure.

### Example: NoSQL Database (MongoDB)

#### Student Document

```
{  
  "_id": 1,  
  "name": "Ravi",  
  "branch": "CSE",  
  "marks": 85,  
  "subjects": ["DBMS", "FSD"],  
  "address": {  
    "city": "Hyderabad",  
    "state": "Telangana"  
  }  
}
```

✓ Fields can be added or removed without affecting other documents.

#### Diagram: SQL vs NoSQL Storage Model

SQL Database	NoSQL Database
-----	-----
Tables	Collections
Rows	Documents
Columns	Fields
Foreign Keys	Embedded Data
Fixed Schema	Flexible Schema

## 2Q. Describe MongoDB data types and JSON document structure with examples.

The name **MongoDB** comes from the word “**humongous**”, meaning **very large or massive**. It reflects the database’s original goal: **to handle huge volumes of data efficiently**, especially for large-scale web applications.

MongoDB is a **document-oriented NoSQL database** that stores data in the form of **documents** rather than tables and rows as in relational databases. MongoDB stores documents internally in **BSON (Binary JSON)** format, which is an extended version of JSON designed for efficiency, speed, and additional data types. Each MongoDB document represents a **self-contained data record**, making MongoDB highly suitable for modern web applications that require **flexible schemas, scalability, and high performance**.

### MongoDB Data Storage Model

MongoDB follows a hierarchical structure:

#### MongoDB Server

└─ **Database**

    └─ **Collection**

        └─ **Document**

            └─ **Fields (Key–Value pairs)**

**Database** → Logical container for collections

**Collection** → Group of documents (similar to tables)

**Document** → Individual record (stored as BSON/JSON)

### JSON Document Structure in MongoDB:

A MongoDB document is a **JSON-like object** composed of **field–value pairs**. Each field has a name (key) and a value of a specific data type.

#### General JSON Document Format

```
{  
  "field1": value1,  
  "field2": value2,  
  "field3": value3  
}
```

MongoDB automatically adds a unique **\_id** field to every document.

#### Example MongoDB Document (Student Collection)

```
{  
  "_id": ObjectId("64fab1234"),  
  "rollNo": 101,  
  "name": "Ravi",  
  "branch": "CSE",  
  "age": 21,  
  "marks": 85,  
  "subjects": ["DBMS", "FSD", "CN"],  
  "address": {  
    "city": "Hyderabad",  
    "state": "Telangana"  
  },  
}
```

```
"isActive": true,  
"admissionDate": ISODate("2023-07-15")  
}
```

This example demonstrates MongoDB's ability to store **structured, semi-structured, and nested data** in a single document.

## MongoDB Data Types :

### 1. String

- Used to store text data
- Most commonly used data type

**Example: "name": "Ravi"**

### 2. Number :

MongoDB supports different numeric formats:

- Integer
- Double
- Long

**Example: "age": 21,  
"marks": 85.5**

### 3. Boolean :

- Stores logical values: true or false

**Example: "isActive": true**

### 4. Array:

- Stores multiple values in a single field
- Values can be of same or different types

**Example:**

**"subjects": ["DBMS", "FSD", "CN"]**

### 5. Embedded Document (Object) :

- A document inside another document
- Used to represent related data

**Example: "address": {  
"city": "Hyderabad",  
"state": "Telangana"  
}**

### 6. ObjectId :

- A **unique identifier** automatically generated by MongoDB
- Acts as the primary key

**Example:**

**"\_id": ObjectId("64fab1234")**

The ObjectId contains:

- Timestamp

- Machine identifier
- Process ID
- Counter

## **7. Date :**

- Stores date and time information
- Useful for auditing and logging

### **Example:**

"admissionDate": ISODate("2023-07-15")

## **8. Null :**

- Represents an empty or missing value

**Example:** "middleName": null

## **9. Binary Data:**

- Used to store images, files, or encrypted data

### **Example:**

"profilePic": BinData(0, "AbCdEf==")

## **10. Regular Expression**

- Used for pattern matching in queries

**Example:** "name": { "\$regex": "^S" }

## **1. Schema-less Design**

- Documents in the same collection can have different fields
- No predefined schema is required

### **Example:**

```
{ "name": "Ravi", "marks": 80 }
{ "name": "Anil", "marks": 85, "grade": "A" }
```

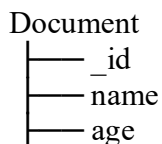
## **2. Self-Describing Documents**

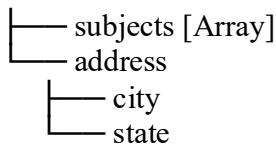
Each document contains both data and structure, unlike relational rows.

## **3. Denormalized Data**

Related data is often stored together using embedded documents and arrays.

### **Diagram: MongoDB JSON Document Structure**





## Advantages of MongoDB JSON Document Model

MongoDB's **JSON/BSON document structure and rich data types** provide a flexible, powerful, and efficient way to store modern application data. The ability to store **nested documents, arrays, and diverse data types** within a single document eliminates complex joins and improves performance.

Therefore, MongoDB is well suited for **full stack development**, especially when integrated with **Node.js**, making it an essential database technology for modern web applications.

## 4Q. Explain MongoDB Data Modeling: Embedded Documents vs References

Data modeling is the process of designing how data is **stored, organized, and related** within a database. Unlike relational databases that rely on **tables, foreign keys, and joins**, MongoDB uses a **document-oriented data model**.

MongoDB data modeling focuses on:

- **How data is accessed**
- **Application usage patterns**
- **Performance and scalability**

MongoDB primarily supports **two data modeling techniques**:

1. **Embedded Documents**
2. **References**

Choosing the correct model is critical for application performance and maintainability.

### MongoDB Data Modeling Philosophy :

MongoDB modeling is **query-driven**, not normalization-driven.

Key principles:

- Data is often **denormalized**
- Joins are avoided
- Related data is stored together whenever possible
- Read performance is prioritized over strict normalization

## 1. Embedded Documents (Embedding Model)

### Definition

An **embedded document** is a document that is stored **inside another document** as a nested object or array.

“Storing related data together within a single document to optimize read performance.”

### Example: Embedded Document Model

#### *Order Collection (Embedded Customer Data)*

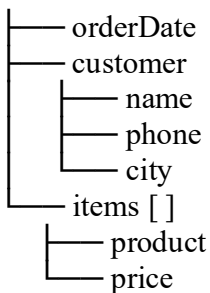
```
{
  "_id": 101,
  "orderDate": "2024-01-10",
  "customer": {
    "name": "Ravi",
    "phone": "9876543210",
    "city": "Hyderabad"
  },
  "items": [
    { "product": "Laptop", "price": 55000 },
    { "product": "Mouse", "price": 800 }
  ]
}
```

Here:

- customer is an **embedded document**
- items is an **array of embedded documents**

Diagram: Embedded Model :

Order Document



### Advantages of Embedded Documents

1. **Faster Read Performance**
  - Entire related data is fetched in a single query.
2. **No Joins Required**
  - MongoDB avoids expensive join operations.
3. **Atomic Operations**
  - Updates to embedded data are atomic at document level.
4. **Simple Query Structure**
  - Data access becomes straightforward.

## When to Use Embedded Documents :

- One-to-few relationships
- Data accessed together frequently
- Data does not grow unbounded
- Strong ownership relationship

## 2. References (Referencing Model)

### Definition

In the **reference model**, related documents are stored in **separate collections** and linked using **ObjectId** references.

“Storing relationships using references when data grows independently or is shared.”

### Example: Reference Model

#### *Customer Collection*

```
{
  "_id": ObjectId("abc123"),
  "name": "Ravi",
  "phone": "9876543210",
  "city": "Hyderabad"
}
```

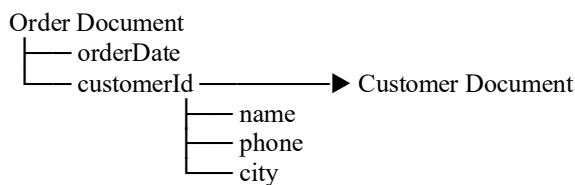
#### *Order Collection*

```
{
  "_id": 101,
  "orderDate": "2024-01-10",
  "customerId": ObjectId("abc123"),
  "totalAmount": 55800
}
```

Here:

- customerId is a **reference** to another collection

### Diagram: Reference Model



### Advantages of References

1. **Reduced Data Duplication**
  - Single copy of shared data.
2. **Better for Large & Growing Data**
  - Suitable for one-to-many and many-to-many relationships.
3. **Easier Updates**
  - Update data in one place.

## Disadvantages of References

1. **Multiple Queries Required**
  - Data retrieval may need multiple database calls.
2. **Slower Read Performance**
  - Compared to embedded model.
3. **More Complex Application Logic**
  - Application must manually resolve references.

## When to Use References :

- One-to-many or many-to-many relationships
- Data shared across multiple documents
- Data grows independently
- Avoid document size limit issues

## Comparison: Embedded Documents vs References

Feature	Embedded Documents	References
Read Performance	High	Moderate
Data Duplication	Yes	No
Query Complexity	Simple	Complex
Joins Required	No	Application-side
Scalability	Limited by document size	High
Use Case	One-to-few	One-to-many

MongoDB data modeling is **application-driven and performance-oriented**, unlike traditional relational modeling.

- **Embedded documents** are ideal for tightly coupled, frequently accessed data.
- **References** are suitable for shared, large, or growing datasets.

## 5Q. Explain how to connect MongoDB with Node.js using MongoClient.

In Full Stack Development, the backend server must communicate with a database to store, retrieve, update, and delete data. MongoDB provides an official **MongoDB Node.js Driver** that allows Node.js applications to interact with MongoDB databases.

**MongoClient** is the primary class used to:

- Establish a connection with MongoDB
- Access databases and collections
- Perform CRUD operations

MongoDB integrates naturally with Node.js because both use **JavaScript and JSON-like data structures**.

### Why MongoClient is Required

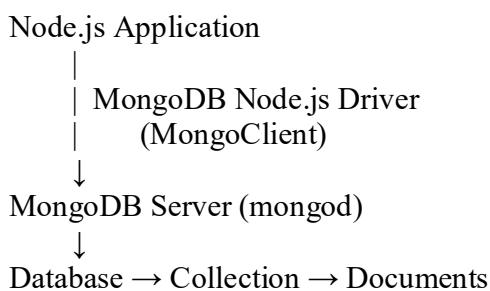
MongoDB runs as a separate database server, while Node.js runs as an application server. MongoClient acts as a **bridge** between:

- Node.js application
- MongoDB server

Using MongoClient, Node.js can:

- Open a connection to MongoDB
- Send database commands
- Receive results asynchronously

### Architecture Diagram :



### Steps to Connect MongoDB with Node.js using MongoClient :

#### Step 1: Install MongoDB Node.js Driver

The MongoDB driver must be installed using npm.

```
npm install mongodb
```

This installs the official MongoDB driver used in Node.js applications.

## Step 2: Import MongoClient Module

In Node.js, the MongoClient class is imported from the mongodb package.

```
const { MongoClient } = require("mongodb");
```

This statement allows the program to use MongoClient for database connection.

## Step 3: Define MongoDB Connection URL

MongoDB uses a connection string (URL) to specify:

- Protocol
- Host
- Port

```
const url = "mongodb://127.0.0.1:27017";
```

- 127.0.0.1 → Localhost
- 27017 → Default MongoDB port

## Step 4: Create MongoClient Object

```
const client = new MongoClient(url);
```

This creates a MongoClient instance that will manage the connection.

## Step 5: Establish Connection to MongoDB

MongoDB connection is asynchronous, so async/await is used.

```
async function connectDB() {  
  await client.connect();  
  console.log("Connected to MongoDB");  
}  
connectDB();
```

Once connect() is successful, Node.js is connected to MongoDB.

## Step 6: Access Database and Collection

MongoDB creates databases and collections automatically when data is inserted.

```
const db = client.db("collegeDB");  
const students = db.collection("student");
```

- collegeDB → Database name
- student → Collection name

## Example: MongoDB Connection using MongoClient

```
const { MongoClient } = require("mongodb");

const url = "mongodb://127.0.0.1:27017";
const client = new MongoClient(url);

async function main() {
  try {
    // Connect to MongoDB
    await client.connect();
    console.log("MongoDB Connected Successfully");

    // Access database and collection
    const db = client.db("collegeDB");
    const students = db.collection("student");

    // Insert a document
    await students.insertOne({
      name: "Ravi",
      marks: 85,
      branch: "CSE"
    });

    console.log("Student record inserted");
  } catch (error) {
    console.error(error);
  } finally {
    // Close connection
    await client.close();
  }
}

main();
```

### Explanation of the Code :

#### 1. MongoClient Creation

- MongoClient object manages the database connection.

#### 2. Asynchronous Connection

- `await client.connect()` ensures the connection is established before operations.

#### 3. Database Selection

- `client.db("collegeDB")` selects or creates the database.

#### 4. Collection Access

- `db.collection("student")` selects or creates the collection.

#### 5. CRUD Operations

- After connection, MongoDB operations can be performed.

## Advantages of Using MongoClient

1. Official MongoDB driver
2. Asynchronous and non-blocking
3. Easy integration with Node.js
4. Supports all MongoDB operations
5. Scalable and efficient

## 6Q. Discuss MongoDB User Management, Authentication and Access Control

In modern web applications, data security is a critical requirement. MongoDB provides a comprehensive **security model** to protect databases from unauthorized access.

MongoDB security is mainly achieved through:

1. **User Management**
2. **Authentication**
3. **Authorization (Access Control using Roles and Privileges)**

These mechanisms ensure that only authorized users can access and perform operations on MongoDB databases.

### 1. MongoDB User Management

#### Definition

User Management in MongoDB refers to the process of **creating, modifying, and managing users** who can access the MongoDB server.

MongoDB users are:

- Created at the **database level**
- Assigned **roles** that define their permissions

#### Types of Users

MongoDB supports:

- **Database users** – access specific databases
- **Administrative users** – manage users, roles, and databases

#### Creating a User

Users are created using the `createUser()` command.

#### *Example: Creating a Database User*

use collegeDB

```
db.createUser({  
  user: "studentUser",
```

```
pwd: "student123",
roles: [
  { role: "readWrite", db: "collegeDB" }
];
});
```

### Explanation:

- user → Username
- pwd → Password
- roles → Permissions assigned to the user

## Managing Users

MongoDB provides commands to manage users:

- db.createUser() – Create a new user
- db.dropUser() – Delete a user
- db.updateUser() – Modify user roles or password
- db.getUsers() – View existing users

## 2. MongoDB Authentication

### Definition

Authentication is the process of **verifying the identity of a user** before allowing access to the database.

MongoDB uses **username and password–based authentication**.

### Enabling Authentication

Authentication is enabled by starting MongoDB with the --auth option or configuring it in mongod.cfg.

security:

authorization: enabled

Once enabled, **all users must authenticate** before accessing MongoDB.

### Authenticating a User

Users must provide credentials while connecting.

### *Example: MongoDB Shell Authentication*

```
mongosh -u studentUser -p student123 --authenticationDatabase collegeDB
```

### Authentication Databases

MongoDB stores user credentials in:

- The database where the user is created
- Or in the admin database (for admin users)

Authentication Mechanism

- MongoDB securely stores hashed passwords
- Credentials are validated before granting access
- Prevents unauthorized database access

3. MongoDB Authorization (Access Control)

Definition

Authorization determines **what actions a user is allowed to perform** after authentication. MongoDB uses **Role-Based Access Control (RBAC)**.

Roles in MongoDB

A **role** is a collection of **privileges** that define allowed actions on resources.

Built-in Roles

MongoDB provides predefined roles such as:

Role	Description
read	Read-only access
readWrite	Read and write access
dbAdmin	Database administration
userAdmin	User management
clusterAdmin	Cluster management
root	Full access

Example: Assigning a Role

```
db.createUser({
  user: "adminUser",
  pwd: "admin123",
  roles: ["dbAdmin"]
});
```

Privileges in MongoDB

Definition

A **privilege** defines:

- **Resource** (database or collection)
- **Actions** (find, insert, update, delete, etc.)

### Example Privilege Structure

```
{
  resource: { db: "collegeDB", collection: "student" },
  actions: ["find", "insert"]
}
```

### Custom Roles

MongoDB allows creation of **custom roles**.

#### Example: Custom Role

```
db.createRole({
  role: "studentRole",
  privileges: [
    {
      resource: { db: "collegeDB", collection: "student" },
      actions: ["find"]
    }
  ],
  roles: []
});
```

### Assigning Custom Role to User

```
db.createUser({
  user: "viewer",
  pwd: "view123",
  roles: ["studentRole"]
});
```

### Security Flow in MongoDB

```
User
↓
Authentication (Username + Password)
↓
Role Assignment
↓
Privileges
↓
Database / Collection Access
```

### Advantages of MongoDB Security Model

1. Fine-grained access control
2. Role-based permissions
3. Secure authentication mechanism
4. Supports enterprise-level security.
5. Prevents unauthorized access.